第5章 逻辑回归与梯度下降

本章学习目标

- 理解逻辑回归的概念；
- 了解梯度下降算法；
- 了解随机梯度下降算法；
- 掌握通过逻辑回归实现手写数字识别的方法。

在机器学习领域，逻辑回归(logistic regression)常被用于解决二分类问题，例如用于预测某个事件发生的可能性：预测某用户购买某件商品的可能性，或者预测病人患有某种疾病的可能性。逻辑回归是广义线性模型的一个特例，虽然被称为回归，但在实际应用中常被用于分类任务。接下来，本章将介绍通过最优化算法训练非线性函数模型并将模型用于解决分类任务的方法。

5.1 逻辑回归与 Sigmoid 函数

5.1.1 逻辑回归简介

机器学习通常由三个要素构成：模型、策略和算法。模型是指假设数据空间的形式(例如线性模型或者条件概率模型)；策略是指模型好坏的判别标准，即优化问题(损失函数)；算法是指优化问题的求解方法(例如梯度下降算法)。

逻辑回归与线性回归都属于广义线性模型。逻辑回归假设因变量 y 服从伯努利分布，而线性回归假设因变量 y 服从高斯分布，逻辑回归与线性回归有许多相似之处。简单来说，逻辑回归模型就是在线性回归的结果中添加了 Sigmoid 函数。逻辑回归模型通过引入 Sigmoid 函数为模型增加了非线性因素，因此可以更轻松地处理 0-1 分类问题。

逻辑回归是在线性回归的基础上加了一个 Sigmoid 函数(非线性)映射，使得逻辑回归成为一个优秀的分类算法。从本质上来说，两者都属于广义线性模型，但它们两个要解决的问题不一样：逻辑回归解决的是分类问题，输出的是离散值；线性回归解决的是回归问题，输出的是连续值。

逻辑回归的求解过程可以大致分为以下三个步骤。

(1) 寻找合适的函数，用来预测输入数据的分类结果。

(2) 构建损失函数，该函数用于表达预测输出与训练数据类别之间的偏差。损失函数通常会计算预测的输出值与实际值的各类差值作为偏差评价标准。

(3) 找到损失函数的最小值(损失值越小表示预测函数的预测越准确)。求解损失函数的最小值通常采用梯度下降法(gradient descent)。二分类问题中一般使用 Sigmoid 函数作为预测分类函数(5.1.2 节将会详细讲解)。

通过 Python 实现逻辑回归的一般流程如下。

(1) 收集数据：进行原始数据的收集。

(2) 预处理数据：由于计算中涉及距离运算,因此需要将原始数据的数据类型转换为数值型(结构化数据格式效果更好)。

(3) 数据分析：采用相应的分析方法对数据进行分析。

(4) 训练模型：训练模型将花费大量的时间资源和计算资源,训练模型的目的在于找到最佳的分类回归系数。

(5) 测试模型：当模型训练完成后,通过测试数据集对模型进行检测,测试模型的预测准确度。

(6) 应用模型：基于之前训练好的模型对这些数值进行简单的回归计算,判定新的未知数据所属的类别。

5.1.2 Sigmoid 函数简介

逻辑回归的目的在于找到能对所有输入数据准确预测的分类函数。以二元分类为例,输出结果只有“0”或“1”两个,即 $y\in\{0,1\}$,阶跃函数曾被用于处理此类问题,但是阶跃函数在 $x=0$ 时位置会发生突变,这个突变在数学上很难处理,即阶跃函数具有不连续、不可导的特点,因此在逻辑回归里引入了 Sigmoid 函数。Sigmoid 函数的表达式为

$$g(z)=\frac{1}{1+e^{-z}}$$

Sigmoid 函数的示意图如图 5.1 所示。

通过 Sigmoid 函数可以将输入数据压缩到区间(0,1)内,得到的结果不是二值输出而是概率值,即当一个 x 发生时,y 被分到 0 或 1 的概率。因此,逻辑回归也可以被看成一种概率估计算法。事实上,最终得到的 y 值是在(0,1)这个区间上的某个值,然后根据事先设定的一个阈值,通常是 0.5(在实际应用中可适当调整该阈值),当 $y>0.5$ 时,就将这个 x 归为 1 类；当 $y<0.5$ 时,将 x 归为 0 类。当 x 为 0 时,Sigmoid 函数值为 0.5；当 x 的值增大时,Sigmoid 函数值将趋近 1；当 x 的值减小时,Sigmoid 函数值将趋近 0。

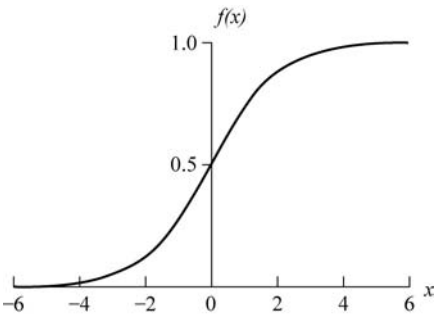

图 5.1 Sigmoid 函数

5.2 梯度下降算法

梯度下降算法是一种常见的最优化算法,该算法通过梯度来求解函数极小值,梯度下降的方向便是调整变量的方向。梯度下降算法是求解无约束优化问题时最常采用的经典方法

之一。机器学习的核心内容就是通过模型自动地“学习”数据，优化模型自身的参数并完成“学习任务”。这个“学习”的过程就是最优化的过程。

在数学中，求解一个函数的最值问题时，往往采用求导的方法，寻找函数导数为 0 的点，进而判断该点是否是该函数的最值点。但是实际应用中，很难求解出函数导数为 0 的解析式，而梯度下降算法的出现解决了这一问题。梯度下降算法通过沿着函数的梯度方向更新参数来寻找函数的极值点。之所以沿着梯度方向更新参数，是因为通常一个函数的梯度方向是该函数的值变化最快的方向。

假设有函数 $y=f(x)$，该函数的一阶导数为 $f'(x)$，梯度算法迭代的方向 d 由 $f'(x)$ 决定。导数 $f'(x)$ 代表 $f(x)$ 在点 x 处的斜率指向函数值变化最快的方向。因此，如果把迭代的每一步沿着当前点的梯度的相反方向移动，则可以得到一个逐步减小的极小化序列，具体如图 5.2 所示。函数 $y=f(x)$ 的极值点为 $f'(x)=0$，通过求解方程 $f'(x)=0$ 的值求得函数的极值点 (x_0,y_0)。

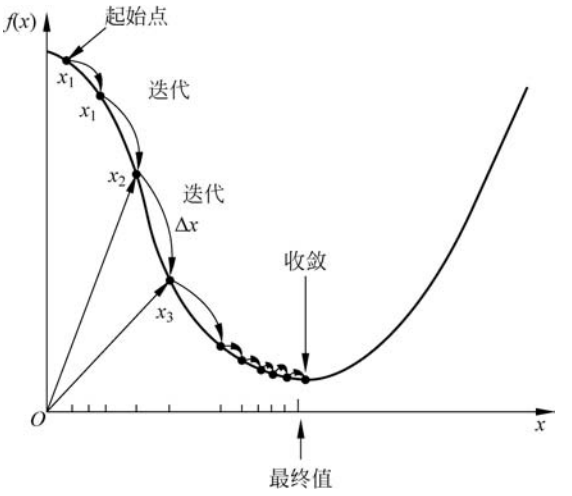

图 5.2　梯度下降算法

在图 5.2 中，梯度下降算法的流程为：先在函数曲线上随机选择一个起始点（图中选择了点 x_1）进行求导。然后，每次迭代沿着梯度相反方向调整 x 的值，使得函数的值向着极小值方向移动，经过多次迭代后最终达到函数极小值点，即 $f'(x)=0$ 处。每次迭代调整 x 的值的幅度称为步长值，由于步长值的设定很难精确到恰巧求得函数的极小值，因此梯度下降算法中所求得的极小值往往是真正的极小值附近的值，真正的极小值很可能处在最后两次迭代值之间。步长值选择会极大地影响梯度下降算法，步长值过小，会增加求得极小值的迭代过程；步长值过大，则很容易错过极小值，最终收敛到错误的点上。

当 $f'(x)=0$ 时，导数将无法提供往哪个方向移动的信息，因此，$f'(x)=0$ 的点称为临界点（critical point）或驻点（stationary point）。一个局部极小点（local minimum）意味着这个点的 $f(x)$ 小于所有邻近点，这意味着，此时无法通过调整 x 的值来进一步减小 $f(x)$ 的值。一个局部极大点（local maximum）意味着这个点对应的 $f(x)$ 的值大于所有邻近点，在此点处不可能通过移动无穷小的步长来进一步增大 $f(x)$ 的值。有些临界点既不是最小点也不是最大点。这些点被称为鞍点（saddle point），如图 5.3 所示。

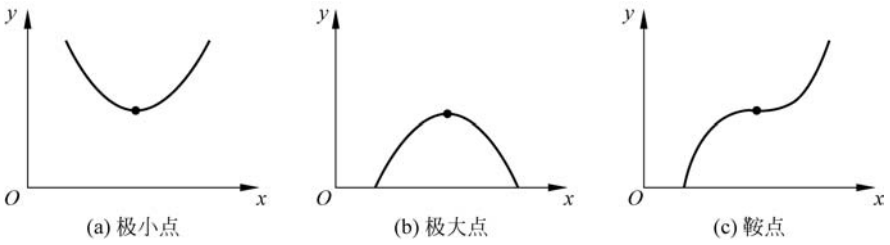

图 5.3　临界点的类型

图中的三个点均为斜率为零的点，被称为临界点。从图 5.3 中可以看出这三个临界点分别对应了以下三种情况。

- 局部极小点，其值低于相邻点。
- 局部极大点，其值高于相邻点。
- 鞍点，同时存在更高和更低的相邻点。

在函数 $f(x)$上处于全局的绝对极小值的点被称为全局最小值点(global minimum)。一个函数可以存在多个局部极小值，但只存在唯一全局极小值(注意是极小值唯一，而不是极小值点唯一)，在实际应用中应避免将不是全局最优的局部极小点误判为全局最小值点。在机器学习领域，优化的函数可能包含多个非全局最优的局部极小值点，或多个被非常平坦的区域包围的鞍点。尤其是在多维的情况下，优化会变得非常困难。因此，在实际情况中通常找到函数 $f(x)$的一个非常小的值近似为全局极小值，即近似最小化，如图 5.4 所示。

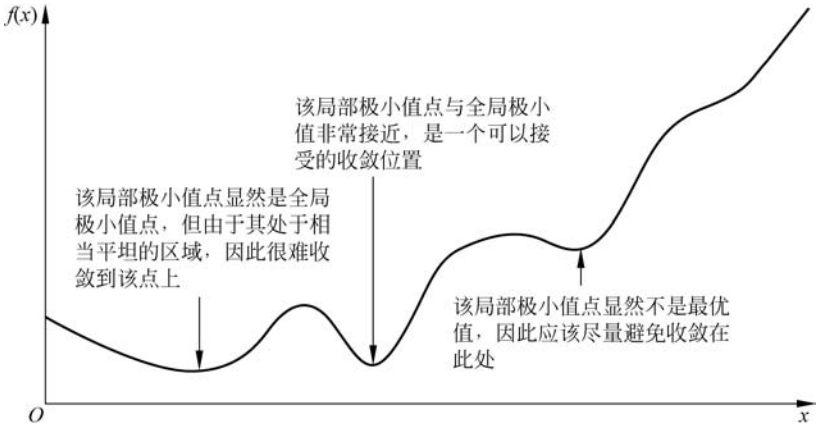

图 5.4　近似最小化

图 5.4 中存在多个局部极小值点或平坦区域，此时，梯度下降算法很难找到全局最小值点。此时可以通过近似最小化操作找到损失函数值的显著极低的点作为全局最小值点。

在机器学习实践中，经常遇到最小化具有多维输入的函数的情况。为了“最小化”多维函数，必须让该函数的输出是一维的(标量)。针对具有多维输入的函数，需要用到偏导数(partial derivative)。偏导数$\frac{\partial}{\partial x_i}f(x)$衡量点 x 处只存在 x_i 增量时函数的变化情况。函数 $f(x)$的导数包含所有偏导数的向量，记作$\nabla_x f(x)$。梯度的第 i 个元素表示函数 $f(x)$关于 x_i 的偏导数。在具有多维输入的函数中，临界点为所有梯度元素都为零的点。

梯度下降算法的伪代码如下所示。

```
将每个回归系数初始化为 1
重复 R 次:
计算整个数据集的梯度
使用 alpha × gradient 更新回归系数的向量
返回回归系数
```

5.2.1 二维坐标系中的梯度下降算法

假设需要通过梯度下降算法求解最小值的目标函数为 $f(x)=x^2+1$。该函数的坐标如图 5.5 所示。

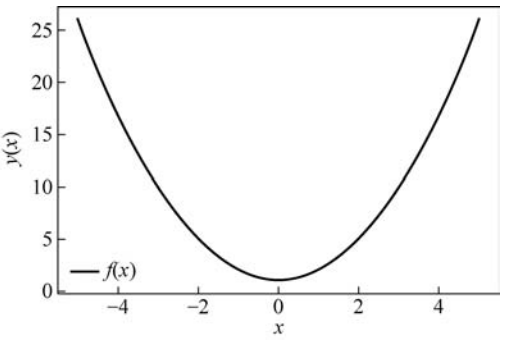

图 5.5 函数坐标

从图 5.5 可以看出,函数的最小值点出现在 $x=0$ 处。接下来,通过一段简单的代码,演示通过 Python 实现梯度下降算法的过程。具体代码如例 5.1 所示。

【例 5.1】 简单的梯度下降算法。

```
1  def func_1d(x):
2      """
3      目标函数
4      :param x: 自变量,标量
5      :return: 因变量,标量
6      """
7      return x ** 2 + 1
8
9  def grad_1d(x):
10     """
11     目标函数的梯度
12     :param x: 自变量,标量
13     :return: 因变量,标量
14     """
15     return x * 2
16
17  def gradient_descent_1d(grad, cur_x = 0.1, learning_rate = 0.1, precision = 0.0001, max_
    iters = 10000):
18     """
19     梯度下降算法
20     :param grad: 目标函数的梯度
21     :param cur_x: 当前 x 值,通过参数可以提供初始值
22     :param learning_rate: 学习率(步长值)
23     :param precision: 设置收敛精度
24     :param max_iters: 最大迭代次数
25     :return: 局部最小值 x*
26     """
27     for i in range(max_iters):
```

```
        grad_cur = grad(cur_x)
        if abs(grad_cur) < precision:
            break                    # 当梯度趋近于 0 时,视为收敛
        cur_x = cur_x - grad_cur * learning_rate
        print("第", i, "次迭代:x 值为 ", cur_x)

    print("局部最小值 x = ", cur_x)
    return cur_x


if __name__ == '__main__':
    gradient_descent_1d(grad_1d, cur_x = 10, learning_rate = 0.1, precision = 0.000001, max_iters = 10000)
```

输出结果如下所示。

```
第 0 次迭代: x 值为   8.0
第 1 次迭代: x 值为   6.4
第 2 次迭代: x 值为   5.12
第 3 次迭代: x 值为   4.096
⋮
第 70 次迭代: x 值为  1.3164036458569655e-06
第 71 次迭代: x 值为  1.0531229166855724e-06
第 72 次迭代: x 值为  8.424983333484579e-07
第 73 次迭代: x 值为  6.739986666787663e-07
第 74 次迭代: x 值为  5.391989333430131e-07
第 75 次迭代: x 值为  4.313591466744105e-07
局部最小值 x = 4.313591466744105e-07
```

从上述结果中不难看出,经过 75 次迭代后,x 的值从初始点 8.0 逐步逼近了最小点 $f(0)$。上述代码中的计算过程便是一个相对简单的梯度下降算法的迭代过程。

5.2.2 三维坐标系中的梯度下降算法

接下来,将 5.2.1 节中的知识推广到三维坐标系中。假设需要通过梯度下降算法求解最小值的目标函数为 $f(x,y)=-\mathrm{e}^{-(x^2+y^2)}$。该函数在三维坐标中的图像如图 5.6 所示。

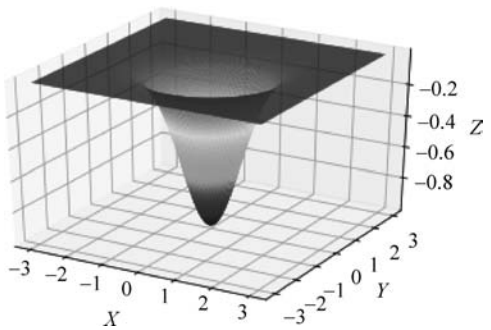

图 5.6 目标函数的三维函数坐标

从图 5.6 中不难看出,函数的最小值点出现在 $x=0$ 处。接下来,通过一段简单的代码,演示通过 Python 实现梯度下降算法的过程。具体代码如例 5.2 所示。

【例 5.2】 三维坐标系中的梯度下降算法。

```
import math
import numpy as np

def func_2d(x):
    """
    目标函数
    :param x: 自变量,二维向量
    :return: 因变量,标量
    """
    return - math.exp( - (x[0] ** 2 + x[1] ** 2))

def grad_2d(x):
    """
    目标函数的梯度
    :param x: 自变量,二维向量
    :return: 因变量,二维向量
    """
    deriv0 = 2 * x[0] * math.exp( - (x[0] ** 2 + x[1] ** 2))
    deriv1 = 2 * x[1] * math.exp( - (x[0] ** 2 + x[1] ** 2))
    return np.array([deriv0, deriv1])

def gradient_descent_2d(grad, cur_x = np.array([0.1, 0.1]), learning_rate = 0.01,
precision = 0.0001, max_iters = 10000):
    """
    二维问题的梯度下降法
    :param grad: 目标函数的梯度
    :param cur_x: 当前 x 值,通过参数可以提供初始值
    :param learning_rate: 学习率,也相当于设置的步长值
    :param precision: 设置收敛精度
    :param max_iters: 最大迭代次数
    :return: 局部最小值 x*
    """
    print(f"{cur_x} 作为初始值开始迭代...")
    for i in range(max_iters):
        grad_cur = grad(cur_x)
        if np.linalg.norm(grad_cur, ord = 2) < precision:
            break  # 当梯度趋近于 0 时,视为收敛
        cur_x = cur_x - grad_cur * learning_rate
        print("第", i, "次迭代:x 值为 ", cur_x)

    print("局部最小值 x = ", cur_x)
    return cur_x

if __name__ == '__main__':
    gradient_descent_2d(grad_2d, cur_x = np.array([1, - 1]), learning_rate = 0.2,
  precision = 0.000001, max_iters = 10000)
```

输出结果如下。

```
[ 1 -1]作为初始值开始迭代...
第 0 次迭代: x 值为   [ 0.94586589 -0.94586589]
第 1 次迭代: x 值为   [ 0.88265443 -0.88265443]
第 2 次迭代: x 值为   [ 0.80832661 -0.80832661]
第 3 次迭代: x 值为   [ 0.72080448 -0.72080448]
第 4 次迭代: x 值为   [ 0.61880589 -0.61880589]
第 5 次迭代: x 值为   [ 0.50372222 -0.50372222]
第 6 次迭代: x 值为   [ 0.3824228 -0.3824228]
⋮
第 31 次迭代: x 值为  [ 1.47596478e-06 -1.47596478e-06]
第 32 次迭代: x 值为  [ 8.85578865e-07 -8.85578865e-07]
第 33 次迭代: x 值为  [ 5.31347319e-07 -5.31347319e-07]
第 34 次迭代: x 值为  [ 3.18808392e-07 -3.18808392e-07]
局部最小值 x = [ 3.18808392e-07 -3.18808392e-07]
```

从上述结果中不难看出,经过多次迭代后,x 的值从初始点 0.945 逐步逼近到了最小点 $f(0)$。上述代码中的计算过程便是一个相对简单的三维坐标系中梯度下降算法的迭代过程。

5.3 通过梯度下降算法找到最佳参数

5.2 节已经介绍了梯度下降算法的相关基础知识,本节通过使用梯度下降算法找到最佳回归系数,即拟合出逻辑回归模型的最佳参数。具体代码如例 5.3 所示。

【例 5.3】 通过梯度下降算法找到逻辑回归模型的最佳参数。

```
1  from numpy import *
2  #加载数据
3  def loadDataSet():
4      dataMat = []; labelMat = []
5      fr = open('testSet.txt')                    #打开文本文件
6      #逐行读取
7      for line in fr.readlines():
8          lineArr = line.strip().split()
9          dataMat.append([1.0, float(lineArr[0]), float(lineArr[1])])
10         labelMat.append(int(lineArr[2]))
11     return dataMat,labelMat                     #返回列表
12
13 def sigmoid(inX):
14     return 1.0/(1+exp(-inX))
15
16 def gradAscent(dataMatIn, classLabels):
17     dataMatrix = mat(dataMatIn)                 #转换为 NumPy 矩阵数据类型,100*3
18     labelMat = mat(classLabels).transpose()     #转置 100*1
19     m,n = shape(dataMatrix)
20     alpha = 0.001
21     maxCycles = 500
22     weights = ones((n,1))  #3*1
23     for k in range(maxCycles):
```

```
24          h = sigmoid(dataMatrix * weights)       #100 * 3 3 * 1 h:100 * 1
25          error = (labelMat - h)                  #100 * 1
26          weights = weights + alpha * dataMatrix.transpose() * error   #最大似然估计
27      return weights
28  dataArraryay,labelMat = loadDataSet()
29  result = gradAscent(dataArraryay,labelMat)
30  print(result)
```

输出结果如下所示。

```
[[ 4.12414349]
 [ 0.48007329]
 [ -0.6168482 ]]
```

在例 5.3 的代码中，首先创建了 loadDataSet()函数，该函数的主要功能是打开文本文件 testSet.txt 并逐行读取。每行前两个值分别是 x_1 和 x_2，第三个值是数据对应的类别标签。为了方便计算，该函数还将 x_0 的值设为 1.0。创建 sigmoid()函数。

梯度下降算法实际由 gradAscent()函数实现，该函数有两个参数。第一个参数是 dataMatIn，该参数为二维 NumPy 数组，每列分别代表不同的特征，每行代表每个训练样本。接下来，获取输入数据并将其转换为 NumPy 矩阵。由于采用的是包含两个特征（x_1 和 x_2）以及第 0 维特征 x_0 的简单数据集，该数据集包含了 100 个样本，dataMathln 中的矩阵维度为 100×3。第二个参数是类别标签，它是一个行向量。为了便于矩阵运算，需要将该行向量转换为列向量（将原向量转置，再将它赋值给 labelMat）。然后得到矩阵大小，并设置梯度下降算法所涉及的参数。

变量 alpha 表示步长值，maxCycles 表示迭代次数。在 for 循环迭代完成后，将返回训练好的回归系数。代码第 24～26 行的运算为矩阵运算。变量 h 是一个列向量，列向量的元素个数等于样本个数（100 个）。对应地，运算 dataMatrix * weights 包含了 300 次乘积运算。

梯度下降算法有着一定局限性，例如，梯度过小时，梯度下降法将难以求解。此外，梯度下降算法通常只用于求解仅有一个局部最优解的目标函数，对于存在多个局部最优解的目标函数，梯度下降法难以求得全局最优解。由于凸函数只存在一个局部最优解，当目标函数是凸函数时，梯度下降法将更容易求得全局最优解。

5.4 决策边界

在二分类问题中，决策边界通常是超曲面，决策边界将基础向量空间划分为两个集合。分类器将决策边界一侧的所有数据点归为同一个类，而将另一侧的所有数据点归为另一个类。本章之前的内容已经介绍了求解回归系数的方法，这些系数确定了不同类别数据之间的分隔线。本节将介绍画出该分隔线的方法，分隔线可以使优化的过程便于理解。具体代码如例 5.4 所示。

【例 5.4】 绘制决策边界。

```
1  def plotBestFit(weights):
2      import matplotlib.pyplot as plt
3      dataMat, labelMat = loadDataSet()     #列表
4      dataArrary = array(dataMat)           #数组 100 * 3
```

```
    n = shape(dataArrary)[0]              #100,对应 100 个训练数据
    xcord1 = []
    ycord1 = []
    xcord2 = []
    ycord2 = []
    for i in range(n):
      if int(labelMat[i]) == 1:           #如果此条训练数据是 1 类
        xcord1.append(dataArrary[i, 1])   #dataArrary[i, 1]为第二列数据,作为作图的横坐标
        ycord1.append(dataArrary[i, 2])   #纵坐标
      else:                               #如果此条训练数据是 0 类
        xcord2.append(dataArrary[i, 1])   #横坐标
        ycord2.append(dataArrary[i, 2])   #纵坐标
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s = 20, c = 'red',marker = 's')
    #s 表示大小,c 表示颜色,marker 表示形状,默认是圆,marker = 's'表示方形
    ax.scatter(xcord2, ycord2, s = 30, c = 'green')
    x = arange( - 3.0, 3.0, 0.1)
    y = ( - weights[0] - weights[1] * x) / weights[2]
    #Sigmoid 函数中,x = 0 是分解条件,对应 y = 0.5
    #映射到特征值:w0 * x0 + w1 * x1 + w2 * x2 = 0
    #其中 x0 = 1,x1 表示横坐标,x2 表示纵坐标
    #解得:x2 = - (w0 + w1 * x1)/w2   ----> y = - (w0 + w1 * x)/w2   即为决策边界
    ax.plot(x, y)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()
dataArraryay,labelMat = loadDataSet()
result = gradAscent(dataArraryay,labelMat)
plotBestFit(result.getA())
```

上述代码通过 Matplotlib 画出了决策边界。需要注意的是,第 23 行代码中的 Sigmoid 函数的值为 0。这是因为 0 是两个分类(类别 1 和类别 0)的分界处。因此,设定 $0=w_0x_0+w_1x_1+w_2x_2$,然后解出 x_1 和 x_2 的关系式(即分隔线的方程,注意 $x_0=1$)。

通过 Matplotlib 绘制的决策边界如图 5.7 所示。

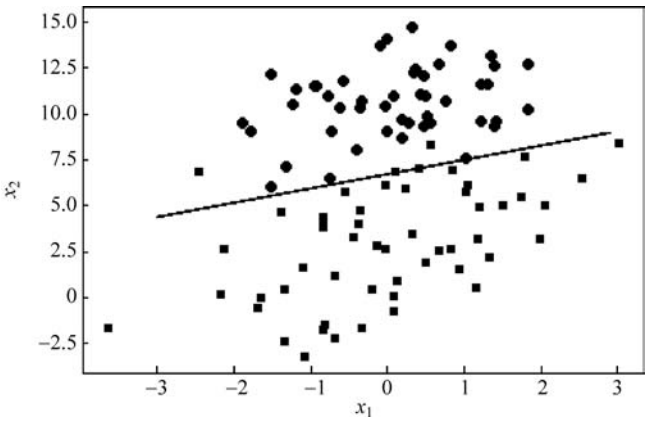

图 5.7　决策边界

从图 5.7 中可以看出，通过例 5.4 的方法划分的数据边界错误划分的数据点少于 6 个。需要指出的是，尽管本例的数据集很小且分布较为简单，但是这个计算方法仍然需要花费大量的计算资源才能得出结果。5.5 节将对该算法进行适当的改进，使算法可以更高效地应用于真实数据集。

5.5 梯度下降算法的改进

在求解机器学习算法的模型参数时，梯度下降算法是最常采用的方法之一，但是这种算法在每次更新回归系数时都需要遍历整个数据集，这使得该算法在处理含有大量样本的数据集时，计算复杂度过高，占用大量的计算资源。随着机器学习技术的发展，衍生出了两种改进算法：批量梯度下降算法（batch gradient descent）和随机梯度下降算法（stochastic gradient descent）。

批量梯度下降算法是梯度下降法最原始的形式，它的具体思路是在更新每一个参数时都使用所有的样本来进行更新。而随机梯度下降算法每次仅用一个样本点来更新回归系数，使得训练速度加快。

5.5.1 批量梯度下降算法

本节将介绍批量梯度下降算法，该算法的优点如下。

- 一次迭代，对所有样本进行计算，此时利用矩阵进行操作，实现了并行。
- 由全数据集确定的方向能够更好地代表样本总体，从而更准确地朝着极值所在的方向。当目标函数为凸函数时，批量梯度下降算法一定能够得到全局最优解。

批量梯度下降算法的缺点如下：当样本数量很大时，每迭代一步都需要对所有样本计算，导致训练效率较低。

接下来将演示实现随机梯度下降算法的方法，具体代码如例 5.5 所示。

【例 5.5】 批量梯度下降算法。

```
1   import  matplotlib.pyplot as plt
2   import random
3   import matplotlib
4
5   #生成数据
6   def data():
7     x = range(10)
8     y = [(2 * s + 4) for s in x]
9     for i in range(10):
10        y[i] = y[i] + random.randint(0,8) - 4
11    return x, y
12
13  #使用梯度下降算法进行训练
14  def diedai(x,y):
15    flag = True
16    a = random.randint(0,5)
```

```
    b = random.randint(0,10)
    m = len(x)
    arf = 0.005 #学习率
    n = 0
    sum1 = 0
    sum2 = 0
    exp = 0.000001
    error0 = 0
    error1 = 0
    while flag:

        for i in range(m):  #计算对应的偏导数
            sum1 += a*x[i]+b-y[i]
            sum2 += (a*x[i]+b-y[i])*x[i]
            error1 += (a*x[i]+b-y[i])**2
        a = a - sum2*arf/m  #对 a,b 进行更新
        b = b - sum1*arf/m

        if abs(error1-error0)<exp: #计算误差
            break
        error0 = error1
        print('a=%f,b=%f,error=%f' % (a, b, error1))

        if n>500:
            #flag = False
            break
        n += 1
        if n % 10 == 0:
            print('第%d次迭代:a=%f,b=%f' % (n, a, b))
    return a,b

#使用最小二乘法计算结果
def calculation(x, y):
    c1 = 0
    c2 = 0
    c3 = 0
    c4 = 0
    n = len(x)
    for i in range(n):
        c1 += x[i]*y[i]
        c2 += x[i]*x[i]
        c3 += x[i]
        c4 += y[i]
    a = (n*c1-c3*c4)/(n*c2-c3*c3)  #利用公式计算 a, b
    b = (c2*c4-c3*c1)/(n*c2-c3*c3)
    return a, b


if __name__ == '__main__':
```

```
    x,y = data()

    a1,b1 = diedai(x,y)
    X1 = range(10)
    Y1 = [(a1 * s + b1) for s in X1]
    print('梯度下降 y = %fX + %f' %(a1,b1))

    a2,b2 = calculation(x,y)
    X2 = range(10)
    Y2 = [(a2 * s + b2) for s in X2]
    print('最小二乘法 y = %fX + %f' %(a2,b2))

    matplotlib.rcParams['font.sans-serif'] = ['SimHei'] #输出中文文本
    plt.scatter(x, y, color = 'red',label = '数据')
    plt.plot(X1, Y1, color = 'blue',label = '梯度下降')
    plt.plot(X2, Y2, color = 'green',label = '最小二乘法')
    plt.legend()
    plt.show()
```

输出结果如下所示。

```
a = 2.955000,b = 4.995000,error = 100.000000
a = 2.911845,b = 4.990205,error = 91.968100
a = 2.870459,b = 4.985607,error = 84.581314
a = 2.830771,b = 4.981197,error = 77.787828
a = 2.792709,b = 4.976968,error = 71.539987
a = 2.756208,b = 4.972912,error = 65.793967
a = 2.721203,b = 4.969023,error = 60.509461
a = 2.687634,b = 4.965293,error = 55.649402
a = 2.655441,b = 4.961716,error = 51.179698
a = 2.624568,b = 4.958285,error = 47.068995
第 10 次迭代:a = 2.624568,b = 4.958285
⋮
a = 1.903025,b = 4.878114,error = 0.000031
第 180 次迭代:a = 1.903025,b = 4.878114
a = 1.903001,b = 4.878111,error = 0.000028
a = 1.902978,b = 4.878109,error = 0.000026
a = 1.902956,b = 4.878106,error = 0.000024
a = 1.902935,b = 4.878104,error = 0.000022
a = 1.902914,b = 4.878102,error = 0.000020
a = 1.902895,b = 4.878099,error = 0.000019
a = 1.902876,b = 4.878097,error = 0.000017
a = 1.902858,b = 4.878095,error = 0.000016
a = 1.902841,b = 4.878093,error = 0.000015
a = 1.902824,b = 4.878092,error = 0.000013
第 190 次迭代:a = 1.902824,b = 4.878092
a = 1.902809,b = 4.878090,error = 0.000012
梯度下降 y = 1.902794X + 4.878088
最小二乘法 y = 1.757576X + 6.290909
```

通过 Matplotlib 绘制的结果如图 5.8 所示。

本书中有很多由 Matplotlib 绘制的图，它们大多只是用于演示数值的变化，没有实际应用的含义，因此图中没有标出 x 和 y。

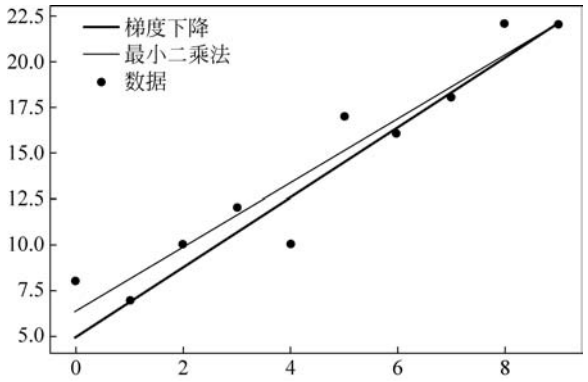

图 5.8　批量梯度下降算法与最小二乘法结果对比

通过图 5.8 可以看出，批量梯度下降算法与最小二乘法的结果存在一定的偏差。事实上每次运行例 5.5 中的代码，所得到的结果都不尽相同。批量梯度下降算法使用了所有的数据集，因此，当数据集较为庞大时，容易出现溢出，或者计算时间非常长的问题。

溢出可以分为两种情况：上溢和下溢。

上溢(overflow)是指大量级的数字被近似为$+\infty$或$-\infty$的情形，这种情况下进一步的运算通常导致这些无限值变为非数字。

下溢(underflow)是指接近 0 的数被四舍五入为 0 时发生的溢出。例如，通常要避免被 0 除或避免取 0 的对数。

由于在机器学习中经常用到概率，而概率通常在 0 和 1 之间，这使得机器学习更容易发生下溢。许多函数在其参数为零而不是一个很小的正数时才会表现出质的不同。例如，通常要避免被 0 除(一些软件环境将在这种情况下抛出异常，有些会返回一个非数字(not-a-number) 的占位符)或避免取 0 的对数(通常被视为$-\infty$)。

针对批量下降算法的缺点，目前发展出了小批量梯度下降算法(mini-batch gradient descent)。小批量梯度下降算法在每次迭代时使用一批自行选择或随机产生的数据，batch 的值可以自主设定，batch 的值越大越近似于批量梯度下降算法，batch 的值越小越近似于随机梯度下降算法。

接下来将演示实现小批量梯度下降算法的方法，具体代码如例 5.6 所示。

【例 5.6】 小批量梯度下降算法。

```
from matplotlib import pyplot as plt
import random

#生成数据
def data():
    x = range(10)
    y = [(3 * i + 2) for i in x]
    for i in range(len(y)):
      y[i] = y[i] + random.randint(0,5) - 3
    return x,y
```

```
11
12  #用小批量梯度下降算法进行迭代
13  def MBGD(x,y):
14    error0 = 0
15    error1 = 0
16    n = 0
17    m = len(x)
18    esp = 1e-6
19    step_size = 0.01              #选择合理的步长
20    a = random.randint(0,10)      #给 a,b 赋初始值
21    b = random.randint(0,10)
22    while True:
23      trainList = []
24      for i in range(5):          #创建随机的批量
25        trainList.append(random.randint(0,m-1))
26
27      for i in range(5):          #对数据进行迭代计算
28        s = trainList[i]
29        sum0 = a*x[s]+b-y[s]
30        sum1 = (a*x[s]+b-y[s])*x[s]
31        error1 = error1+(a*x[s]+b-y[s])**2
32      a = a - sum1*step_size/m
33      b = b - sum0*step_size/m
34      print('a=%f,b=%f,error=%f'%(a,b,error1))
35
36      if error1-error0<esp:
37        break
38      if n % 10 == 0:
39        print('第%d次迭代:a=%f,b=%f' % (n, a, b))
40    return a,b
41      n = n+1
42      if n>500:
43        break
44    return a, b
45  if __name__ == '__main__':
46    x,y = data()
47    a,b = MBGD(x,y)
48    X = range(len(x))
49    Y = [(a*i+b) for i in X]
50
51    plt.scatter(x,y,color='red')
52    plt.plot(X,Y,color='blue')
53    plt.show()
```

输出结果如下所示。

```
a=2.154064,b=6.952813,error=2513.573043
a=2.145542,b=6.948552,error=2564.211560
a=2.132161,b=6.945875,error=2596.145841
```

```
a = 2.156136,b = 6.948872,error = 2649.682594
a = 2.156136,b = 6.943923,error = 2704.802461
a = 2.156136,b = 6.938979,error = 2735.293826
a = 2.156881,b = 6.939104,error = 2787.233266
a = 2.154785,b = 6.937008,error = 2825.086867
a = 2.156560,b = 6.937452,error = 2896.440981
a = 2.156560,b = 6.932514,error = 2940.515715
第 60 次迭代:a = 2.156560,b = 6.932514
⋮
a = 3.381994,b = -0.069075,error = 43232.930272
a = 3.381763,b = -0.069152,error = 43248.989580
a = 3.391541,b = -0.067755,error = 43284.226930
a = 3.391541,b = -0.064688,error = 43297.423451
a = 3.383000,b = -0.065755,error = 43312.519837
a = 3.343569,b = -0.070137,error = 43354.752452
a = 3.345330,b = -0.069784,error = 43368.700294
a = 3.348084,b = -0.069096,error = 43383.915147
a = 3.350791,b = -0.068419,error = 43401.399767
a = 3.353452,b = -0.067754,error = 43408.398876
第 500 次迭代:a = 3.353452,b = -0.067754
a = 3.364607,b = -0.066160,error = 43441.692654
```

通过 Matplotlib 绘制的小批量梯度下降算法结果如图 5.9 所示。

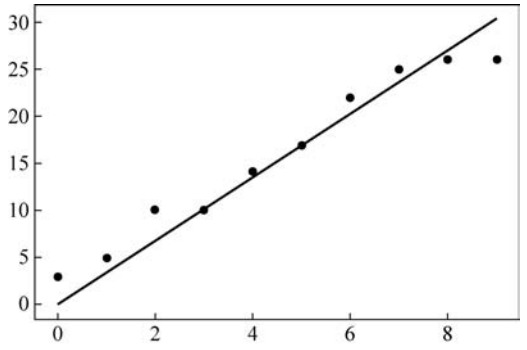

图 5.9　小批量梯度下降算法

读者可以尝试对 batch 的大小进行调整，然后观察 batch 的大小对模型训练效果的影响。

5.5.2　随机梯度下降算法

随机梯度下降算法缓解了梯度下降算法计算量过大的问题，该算法每次迭代使用的只是数据集中的一部分数据，因此不需要像批量梯度下降算法一样每次对整个数据集进行迭代。

随机梯度下降算法的优点：由于随机梯度下降算法在每轮迭代中仅随机优化某一条训练数据上的损失函数，这使得每一轮参数的更新速度大大加快。

随机梯度下降算法的缺点如下所示。

- 准确度有所下降。即使在目标函数为强凸函数的情况下，随机梯度下降算法也无法做到线性收敛。
- 容易陷入局部最优解。这是因为单个样本往往无法代表全体样本的趋势。
- 不易于并行实现。

接下来将演示实现随机梯度下降算法的方法，具体代码如例 5.7 所示。

【例 5.7】 随机梯度下降算法。

```
from matplotlib import pyplot as plt
import  random

#生成数据
def data():
  x = range(10)
  y = [(2 * i + 4) for i in x]
  for i in range(10):
    y[i] = y[i] + random.randint(0,8) - 4
  return x,y

#使用随机梯度下降训练
def SGD(x,y):
  error0 = 0
  step_size = 0.001
  esp = 1e-6
  #a = random.randint(0,4)
  #b = random.randint(0,8)
  a = 1.2  #将给 a,b 随机赋初始值
  b = 3.5
  m = len(x)
  n = 0
  while True:
    i = random.randint(0,m-1)
    print(i)
    sum0 = a * x[i] + b - y[i]
    sum1 = (a * x[i] + b - y[i]) * x[i]
    error1 = (a * x[i] + b - y[i]) ** 2  #计算模型和结果的误差

    a = a - sum1 * step_size
    b = b - sum0 * step_size
    print('a = %f,b = %f,error = %f' %(a,b,error1))

    if abs(error1 - error0)< esp:  #误差很小,可以终止迭代
      break
    error0 = error1
    n = n + 1
    if n % 20 == 0:
      print('第 %d 次迭代' % n)
    if (n > 500):
      break
  return a,b
```

```
43  if __name__ == '__main__':
44    x,y = data()
45    a,b = SGD(x,y)
46    X = range(10)
47    Y = [(a * i + b) for i in X]
48
49    plt.scatter(x,y,color = 'red')
50    plt.plot(X,Y)
51    plt.show()
```

输出结果如下所示。

```
第 10 次迭代
8
a = 1.529368,b = 3.553942,error = 0.045153
1
a = 1.529160,b = 3.553734,error = 4.340184
2
a = 1.530238,b = 3.554273,error = 29.029960
2
a = 1.531315,b = 3.554811,error = 29.000938
2
a = 1.532391,b = 3.555350,error = 28.971944
2
a = 1.533467,b = 3.555888,error = 28.942979
3
a = 1.533720,b = 3.555972,error = 0.711848
0
a = 1.533720,b = 3.555716,error = 6.532993
8
a = 1.533860,b = 3.555734,error = 0.030458
7
a = 1.538555,b = 3.556405,error = 44.987159.
⋮
第 370 次迭代
4
a = 1.798008,b = 3.602848,error = 17.743322
6
a = 1.801373,b = 3.603409,error = 31.462071
9
a = 1.799739,b = 3.603227,error = 3.297009
2
a = 1.800698,b = 3.603707,error = 23.014038
2
a = 1.801657,b = 3.604187,error = 22.991029
3
a = 1.801655,b = 3.604186,error = 0.000084
3
a = 1.801652,b = 3.604185,error = 0.000084
```

通过 Matplotlib 绘制的随机梯度下降算法结果如图 5.10 所示。

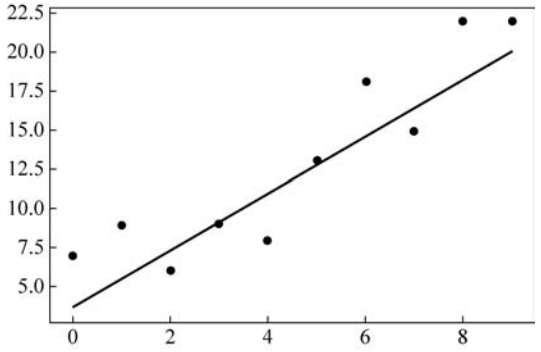

图 5.10　随机梯度下降算法

从图 5.10 可以看出，随机梯度下降算法也得到了较为理想的结果。但是，随机梯度下降算法也存在明显缺陷：由于每次迭代只用到了一组数据，因此，该算法受噪声、离群点和异常值的影响非常大。

到目前为止，本书已经介绍了常见的逻辑回归优化算法的内容和实现方法，感兴趣的读者可以自行研究本书未详细讲解的其他相关算法。值得注意的是，算法中各项参数并不总是一成不变的，读者在实际操作中可以根据不同的数据集对参数进行调整，并观察调整后的效果。

5.6　本 章 小 结

通过本章的学习，可以较为深入地了解逻辑回归算法的相关概念和实现方法。逻辑回归的目的是寻找一个非线性函数 Sigmoid 的最佳拟合参数，求解过程可以由最优化算法来完成。在最优化算法中，最常用的就是梯度下降算法，而梯度下降算法又可以简化为随机梯度下降算法。随机梯度下降算法与梯度下降算法效果相近，但占用的计算资源更少。随机梯度下降算法属于在线算法，它可以在新数据到来时就完成参数更新，而不需要重新读取整个数据集来进行批处理运算。

5.7　习　　题

1. 填空题

（1）逻辑回归与线性回归都属于广义_____模型。

（2）逻辑回归假设因变量服从_____分布。

（3）梯度下降算法通过沿着函数的_____方向更新参数来寻找函数的极小值点。

（4）当目标函数为凸函数时，通过梯度下降法将更_____求得全局最优解。

（5）与随机梯度下降算法相比，_____算法每迭代一步都需要对所有样本计算，导致训练效率较低。

2. 选择题

（1）通过梯度下降算法（　　）准确地找到具体的极值点。

A. 总是可以

B. 通常无法

C. 完全不可能

D. 通常不能快速地找到,但只要模型训练的时间足够长就一定可以

(2) 下列不属于使用增加步长值可能产生的影响的是(　　)。

A. 更容易错过最值点　　B. 学习率降低

C. 更容易出现模型无法收敛的情况　　D. 占用更多的计算资源

(3) 在 NumPy 中,可以用于将数据转换为矩阵的是(　　)。

A. mat()　　B. abs()　　C. dataMatrix()　　D. array()

3. 思考题

(1) 简述梯度下降算法可能存在的局限性。

(2) 简要概括批量梯度下降算法和随机梯度下降算法的优缺点。